

# Computer Systems

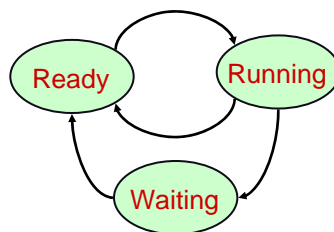
## Topics

- I/O-bound vs. CPU-bound Processes
- Scheduling Algorithms

## The Context

One CPU, multiple processes ...

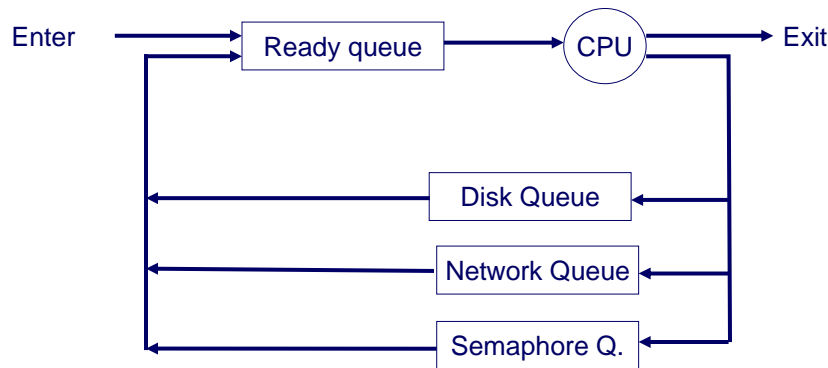
Processes can be in one of the following states:



How does the OS keep track of process states?

- OS maintains a queue of processes ready to run
- OS maintains queues of processes waiting for an event, one queue per event

## OS Queuing Model



Processes enter and leave the system

- 3 -

## CPU Scheduling

Question: Any process in the pool of ready processes is ready to run. Which one to pick to run next?

### CPU scheduling

- Selecting a new process to run from among the pool of Ready processes competing for the CPU
- Basis of multi-programmed OS

### Preemptive scheduling

- running process may be interrupted and moved to Ready queue

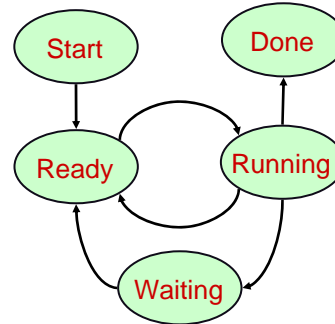
### Non-preemptive scheduling:

- once a process is in Running state, it continues to execute until it terminates or blocks for I/O or system service

- 4 -

## When Does a Scheduler Take Decisions?

1. Running - Waiting
2. Running - Done
3. Waiting - Ready
4. Running - Ready



Scheduling under 1 and 4:

- nonpreemptive scheduling

Scheduling under 2 and 3:

- preemptive scheduling

- 5 -

## What Are We Trying to Optimize?

### System-oriented metrics:

- CPU utilization: percentage of time the processor is busy
- Throughput: number of processes completed per unit of time

### User-oriented metrics:

- Turnaround time: interval of time between submission and termination (including any waiting time). Appropriate for background jobs
- Response time: for interactive jobs, time from the submission of a request until the response begins to be received
- Waiting time: sum of the periods spent in the Ready queue

- 6 -

## Scheduling Criteria

### Maximize

- CPU utilization
- Throughput

### Minimize

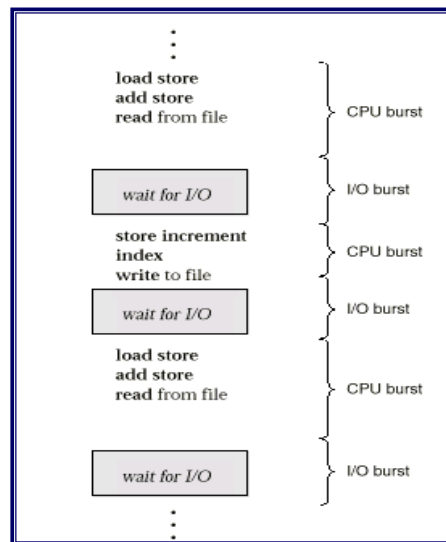
- Turnaround time
- Waiting time
- Response time

### Problem: mutually exclusive objectives

- No one best way
- Waiting / response time vs throughput conflicts

- 7 -

## Alternating CPU and I/O Bursts



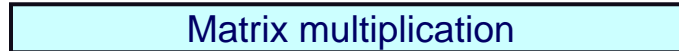
- 8 -

## Process Behavior

Observed property of processes

- alternate between CPU execution and I/O wait

**CPU-bound job:** little I/O, long CPU bursts



**I/O-bound job:** lots of I/O, short CPU bursts



Problem: don't know the bound type before running

An underlying assumption:

- response time most important for I/O bound processes

- 9 -

## Scheduling Algorithms

Next we take a look at four scheduling algorithms and their amusing behavior

1. First Come First Served – FCFS
2. Round Robin – RR
3. Shortest Job First – SJF
4. Priority Scheduling

Scheduling very ad hoc. “Try and See”

- 10 -

## First Come First Served (FCFS or FIFO)

Simplest scheduling algorithm:

- Run jobs in order that they arrive
- Uniprogramming: run until done (non-preemptive)
- Multiprogramming: put job at back of queue when blocks on I/O (we'll assume this)

Advantage: simple

Disadvantages: ???

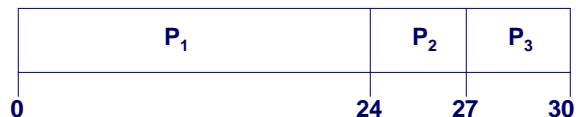
- 11 -

## FCFS Scheduling Example (1)

Example:	<u>Process</u>	<u>Burst Time</u>
	$P_1$	24
	$P_2$	3
	$P_3$	3

Suppose that the processes get created in the order:  $P_1, P_2, P_3$

The chart for the schedule is:



Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$

Average waiting time:  $(0 + 24 + 27)/3 = 17$

Average turnaround time:  $(24+27+30)/3 = 27$

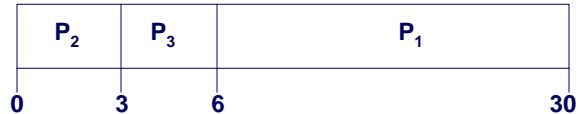
- 12 -

## FCFS Scheduling Example (2)

Suppose that the processes arrive in the order

$$P_2, P_3, P_1$$

The chart for the schedule is:



Waiting time for  $P_1 = 6$ ;  $P_2 = 0$ ;  $P_3 = 3$

Average waiting time:  $(6 + 0 + 3)/3 = 3$

Average turnaround time:  $(30+3+6)/3 = 13$

Much better than previous case

- 13 -

## FCFS Disadvantages

Performance depends on arrival time of processes

*Convoy effect*: short process behind long process

Example: one CPU bound process, many I/O bound

- CPU bound runs (I/O devices idle)
- CPU bound blocks
- I/O bound process(es) run, quickly block on I/O
- CPU bound runs again
- I/O completes
- CPU bound still runs while I/O devices idle (continues...)

Result:

- long periods where no I/O requests issued, and CPU held
- poor I/O device utilization

- 14 -

# Round Robin (RR)

Add a timer

Solution to job monopolizing CPU? Interrupt it.

- Run process for some “time quantum” (time slice)
- When time is up or process blocks, move it to the back of the ready queue
- Most systems do some flavor of this

Advantages:

- Fair allocation of CPU across jobs
- Low average waiting time when job lengths vary:
- Assume 3 processes,  $P_1(100)$ ,  $P_2(2)$ ,  $P_3(1)$  and time quantum = 1

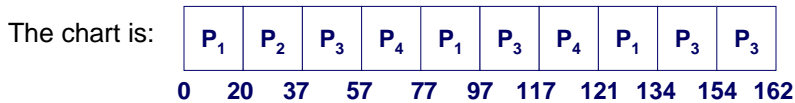


- What is the average waiting time and completion time?

- 15 -

# RR Example: Time Quantum = 20

Process	Burst Time
$P_1$	53
$P_2$	17
$P_3$	68
$P_4$	24



Average waiting time = ?

Average completion time = ?

- 16 -



## RR Time Slice Tradeoffs (2)

Timeslice frequently set to ~100 milliseconds

Context switches typically cost < 1 millisecond

- context switching is usually negligible (< 1% per timeslice in above example) unless you context switch too frequently and lose all productivity.

- 19 -

## Shortest-Job-First SJF

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

### Nonpreemptive

- Once CPU given to the process it cannot be preempted until completes its CPU burst.

### Preemptive

- If a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)

- 20 -

## Example of Non-Preemptive SJF (1)

3 processes,  $P_1(100)$ ,  $P_2(2)$ ,  $P_3(1)$



- Average completion =  $(1+3+103) / 3 = \sim 35$  (vs  $\sim 101$  for FCFS)

Provably optimal

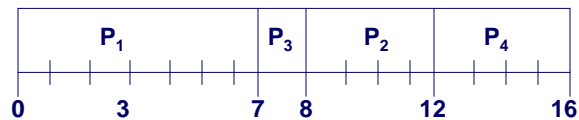
- Moving shorter process before longer process improves waiting time of short process more than harms waiting time for long process

- 21 -

## Example of Non-Preemptive SJF (2)

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	7
$P_2$	2	4
$P_3$	4	1
$P_4$	5	4

STCF (non-preemptive)



Average waiting time =  $(0 + 6 + 3 + 7)/4 = 4$

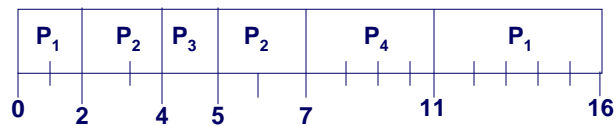
Average completion time = ?

- 22 -

## Example of Preemptive SJF

Process	Arrival Time	Burst Time
$P_1$	0	7
$P_2$	2	4
$P_3$	4	1
$P_4$	5	4

STCF (preemptive)



Average waiting time =  $(9 + 1 + 0 + 2)/4 = 3$

Average completion time = ?

- 23 -

## STCF is Optimal but Unfair

Gives minimum average waiting time

Long-running jobs may starve if too many short jobs

Difficult to implement

- how do you know how long a process takes ?

Option1

- have the user tell us ... if they lie, kill the process
- not so useful in practice

Option 2

- use the past to predict the future

- 24 -

## Use the Past to Predict the Future ...

Use the past to predict the future #1:

- Long running job will probably take a long time more



Use the past to predict the future #2:

- View job as sequence of alternating CPU and I/O jobs



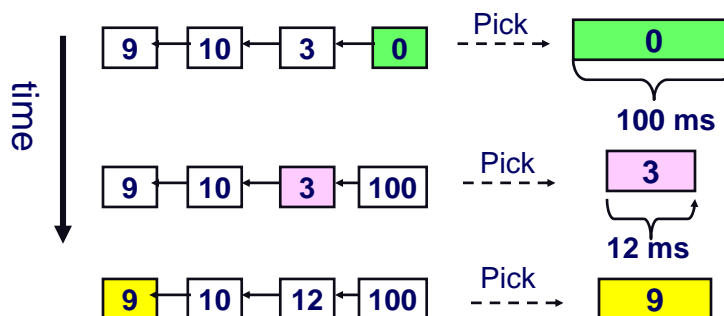
- If previous CPU pieces in the sequence have run quickly, future ones will too (usually)

- 25 -

## Use the Past to Predict the Future ...

Example: Predict length of current CPU burst using length of previous burst

- Record length of previous burst (0 when just created)
- At scheduling event (unblock, block, exit, ...) pick the smallest "past run length" off of Ready queue



- 26 -

## Approximate STCF – Exp. Average

~STCF - exponential averaging

1.  $t_n$  = actual length of  $n^{\text{th}}$  CPU burst
2.  $\tau_{n+1}$  = predicted value for the next CPU burst
3.  $\alpha, 0 \leq \alpha \leq 1$
4. Define:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n.$$

$\alpha = 0$

- $\tau_{n+1} = \tau_n$
- Recent history does not count.

$\alpha = 1$

- $\tau_{n+1} = t_n$
- Only the actual last CPU burst counts

- 27 -

## Exponential Average (2)

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n.$$

If we expand the formula, we get:

$$\begin{aligned} \tau_{n+1} = & \alpha t_n + (1 - \alpha) \alpha t_{n-1} + (1 - \alpha)^2 \alpha t_{n-2} \dots \\ & + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ & + (1 - \alpha)^{n+1} \tau_0 \end{aligned}$$

Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor (recent past counts more)

- 28 -

## Priority Scheduling

SJF, SRTF are special cases of a more general priority scheduling algorithm

Each process has a priority

- Run highest priority ready process in system
- Round robin among processes of equal priority
- Convention: small integer = high priority

Most systems use some variant of this

Common use: couple priority to process characteristic

- Fight starvation? Increase priority of processes waiting in the system for a long time
- Keep I/O busy? Increase priority of processes that often block on I/O

- 29 -

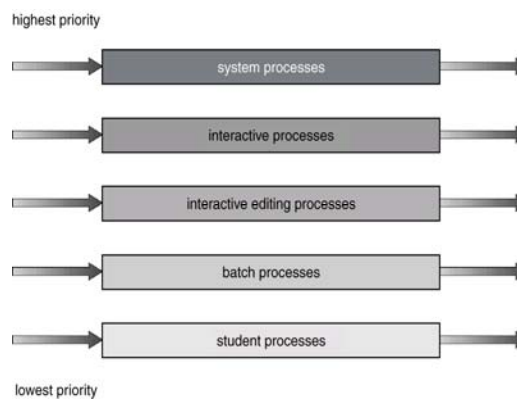
## Multilevel Queue

Ready queue is partitioned into separate queues

Each queue has its own scheduling algorithm

- RR, FCFS

Example:



- 30 -

## Multilevel Feedback Queue

Allows a process to move between the various queues

Look at ~STCF:

- Faster for a longer time slice
- So grow the time slice for a CPU bound process, but decrease its priority to prevent starvation of others

Idea: Separate processes with different CPU bursts

- 31 -

## Multilevel Feedback Queue Ex. (1)

Process created:

- Give high priority and short time slice

If process uses up the time slice without blocking:

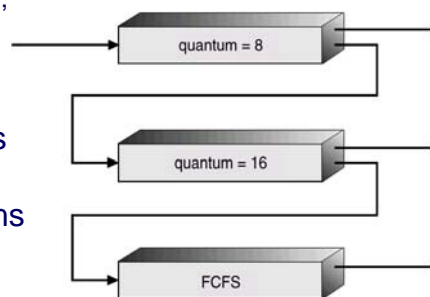
- $priority = priority + 1$  (lower its priority)
- $time\_slice = time\_slice * 2;$

Three queues:

Q0 – time quantum 8 ms

Q1 – time quantum 16 ms

Q2 – FCFS



- 32 -

## Multilevel Feedback Queue Ex. (2)

Three queues:  $Q_0$  (8 ms),  $Q_1$  (16 ms),  $Q_2$  (FCFS)

### Scheduling

- A new process enters queue  $Q_0$  which is served FCFS. When it gains CPU, the process receives 8 milliseconds. If it does not finish in 8 milliseconds, the process is moved to queue  $Q_1$ .
- A  $Q_1$  process is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue  $Q_2$ .
- A  $Q_2$  process is served FCFS

Attacks both efficiency and response time problems

- Efficiency: long time quanta = low switching overhead
- Response time: quickly run after becoming unblocked

- 33 -

## Some problems

A user can insert I/O just to keep priority high

Can't low priority processes starve?

- ad hoc: when skipped over, increase priority

What about when past doesn't predict future?

- For instance, a CPU bound process switches to I/O bound
- want past predictions to "age" and count less towards current view of the world

Windows 2000

- 32 priority levels
- If a running process receives an interrupt, priority is lowered
- If a process unblocks from a waiting queue, priority increases

- 34 -

## Multilevel Feedback Queue

Scheduler defined by the following parameters:

- number of queues
- scheduling algorithms for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process
- method used to determine which queue a process will enter when that process needs service

- 35 -

## Traditional UNIX scheduling (1)

Uses multilevel feedback queues

128 priorities possible (0-127)

1 Round Robin queue per priority

Every scheduling event, the scheduler

- picks the non-empty queue of highest priority
- runs processes in round-robin

Scheduling events:

- Clock interrupt
- Process does a system call
- Process gives up CPU, e.g. to do I/O

- 36 -

## Traditional UNIX Scheduling (2)

All processes assigned a baseline priority based on the type and current execution status:

- swapper 0
- waiting for disk 20
- waiting for lock 35
- user-mode execution 50

At scheduling events, priorities are adjusted based on the amount of CPU used, the current load, and how long the process has been waiting

Most processes are not running, so lots of computing shortcuts are used when computing new priorities

- 37 -

## Linux - Lottery Scheduling

Problem: this whole priority thing is really ad-hoc

- how to ensure that jobs will be equally penalized under load?

Lottery scheduling

- give each process some number of tickets
- at each scheduling event, randomly pick a ticket
- run winning process
- to give a process  $n\%$  of CPU, give it  $\text{total\_tickets} * n\%$

How to use?

- approximate priority: low priority, give few tickets, high priority, give many
- approximates STCF: give short processes more tickets, long jobs fewer. If job has at least 1, won't starve

Linux uses lottery scheduling

- 38 -

## Summary

FIFO: + simple

- short jobs can get stuck behind long ones; poor I/O

RR: + better for short jobs

- poor when jobs are the same length

SJF: + optimal

- hard to predict the future
- unfair to long running jobs

Multi-level queues: Priority + RR

- + approximate STCF
- still unfair to long running jobs